운영체제 8팀 최종 보고서

20193037 이현종

20192636 정재학

20192632 전광현

20192576 김민우

2024.06.12

Table of contents

<Introduction>

MiniOS 프로젝트는 운영체제(OS)의 기본 원리와 설계 방법을 학습하고 실습하는 프로젝트이다.

프로젝트의 목적은 운영체제에 대한 기본적인 개념 이해를 바탕으로 프로세스 관리, 메모리 관리, 파일 시스템 등 운영 체제의 핵심 개념을 직접 구현해보며 익히는 과정에 있다. 우리는 운영체제의 기본 요소인 shell command 부터 system call을 구현하고, CPU 스케쥴링과 파일시스템 같은 OS의 service 들을 구현 하는 것을 목표로 잡았다.

최종적으로, linux 환경 miniOS에서는 shell command들을 구현하여 test까지 진행하였고, 그리고 운영체제의 자원 관리 부분에 초점을 맞춰 더욱 깊이있는 프로젝트 진행을 위해 이용한 xv6에서는 더 나아가서 system call과 CPU scheduling을 3가지 방식으로 구현 하였고 기본적인 file system까지 구현 하였다.

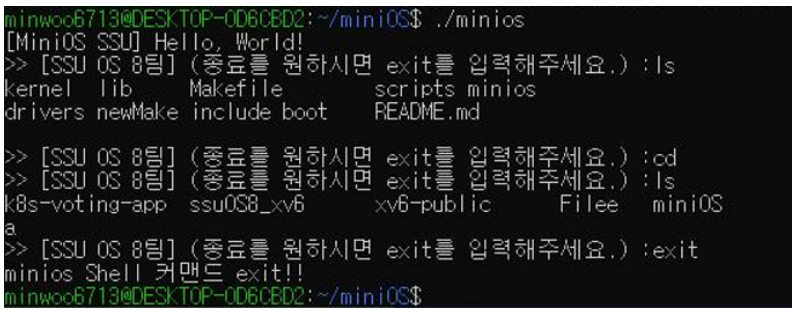
<Body>

1. mimiOS

1) miniOS shell command 구현

쉘 커맨드는 터미널에서 직접 실행되는 단일 명령어로, 파일 및 디렉토리 관리, 프로세스 관리, 시스템 모니터링 등 다양한 작업을 수행할 수 있다. 쉘은 사용자와 커널 사이의 인터페이스 역할을 하기에, 기본적인 운영 체제를 구현하기 위해 필요한 쉘 커맨드 구현에 초점을 맞추어 작업을 시작하였다.

- ls, cd, exit 커맨드 구현



이후 나머지 mkdir, cp, rm, mv, rmdir, cat 커맨드들을 만들었으나 문제점이 발생하였는데, 명령을 입력해도 공백란 자체를 하나의 문자열로 인식하여 원하는 동작의 커맨드를 실행하지 않는 문제였다. 사용자가 입력한 명령어를 해석하고, 명령어에 따른 인자에 맞추어 적절한 동작을 수행하도록 명령어를 파싱하는 코드를 추가로 구현하였다.

```
SYMBOL parse(int *waitpid, BOOLEAN makepipe, int *pipefdp)
{
    SYMBOL symbol, term;
    int argc, sourcefd, destfd;
    int pid, pipefd[2];
    char *argv[MAXARG+1], sourcefile[MAXFNAME];
    char destfile[MAXFNAME];
    char word[MAXWORD];
    BOOLEAN append;

    argc = 0;
    sourcefd = 0;
    destfd = 1;


    while (TRUE)
    {
        switch (symbol = getsymbol(word))
        {
            case S_WORD :
                if(argc == MAXARG)
                {
                    fprintf(stderr, "Too many args.\n");
                    break;
                }
                argv[argc] = (char *) malloc(strlen(word)+1);
```

명령어 파싱의 로직은 입력을 term으로 받아 심볼을 활용하여 switch문을 통해 각 심볼을 처리하고, 일반 단어와 re-direction, 파이프라인을 통해 백그라운드 실행으로 커맨드를 실행할 수 있도록 구성하였다.

parsing 로직을 추가한 뒤 나머지 Shell command들을 실행한 결과가 나오는 것을 확인할 수 있다. 아래는 mkdir 커맨드의 결과이다.

```
>> [SSU OS 8팀] (종료를 원하시면 exit를 입력해주세요.) :mkdir newproject
>> [SSU OS 8팀] (종료를 원하시면 exit를 입력해주세요.) :ls
kernel  lib     Makefile        scripts minios
newproject      drivers newMake include boot
README.md
```

2) About miniOS in linux

처음 miniOS를 develop할 수 있는 방법을 찾아봤을 때 쉘 커맨드 구현, 프로세스 스케줄링, 메모리 관리, 파일 시스템 등 직접적인 하드웨어 제어와 리소스 관리가 필요하다고 생각되었다. 그러나 miniOS의 초기 형태를 Linux에서 실행할 경우, 별도의 운영 체제로서 독립적으로 하드웨어 자원을 관리하지 않는다. 만약 단순히 사용자 공간 프로그램으로 miniOS를 실행한다면(예: miniOS 기능을 시뮬레이션하는 애플리케이션 실행), 이는 Ubuntu의 Linux 커널 위에서 동작하며, 리눅스 커널의 프로세스 스케줄링, 메모리 관리, 드라이버 등의 서비스를 그대로 사용하게 된다. 이 경우 miniOS는 기본적으로 linux 프로세스 중 하나로 실행된다. 따라서 프로젝트 진행을 위해서 운영체제에서 프로세스 스케줄링, 파일시스템, 네트워킹 등의 기능을 직접 구축할 수 있는 환경이 필요시 되었다.

2. shell program & system call of xv6

1) About xv6

linux 위에서 실행되는 miniOS를 대체하여 원하는 환경을 구현할 수 있는 방안을 찾아보던 중 교육적 목적으로 설계된 xv6 에뮬레이터를 찾게 되었다. 앞서 설명한 바와 같이 linux 환경 위에서 실행되는 miniOS의 경우, 부트로딩과 스케줄링, 메모리 관리가 최적화 되어있는 환경에서 동작한다. 반면 xv6는 이미 부트로더, 커널, 유저 프로그램을 포함하는 운영체제를 제공하고 있지만 사용자가 구현 방식에 직접 관여하고 수정하기에 적절한 환경을 제공한다.

따라서 스케줄링 등의 하드웨어 리소스 관리에 대한 핵심 개념을 유저 영역과 커널 영역 사이에서 적용해보기 위하여 QEMU 에뮬레이터 환경에서 운영체제의 기능을 구현하는 것으로 목표로 설정하였다. 기존에 만든 miniOS에서의 쉘 커맨드들을 이식하는 대신 xv6에 기본적으로 구현 되어있고 기능이 동일한 쉘 커맨드들을 그대로 이용하기로 결정하였다. 이후 xv6에 내장되어 있는 커맨드들을 사용하되 각 커맨드를 분석하였다.

2) xv6 shell command 분석

우리가 만든 miniOS 상의 쉘 커맨드와 다르게 어떤 부분이 다른지 체크하였다. 가장 기본적인 차이로는, 커널 영역과 유저 영역에서의 시스템 호출을 통한 처리 과정이었다.

-user program : ‘user’ 디렉토리에 ‘ls’, ‘mkdir’같은 명령어 파일이 포함되어 있고, 이들은 시스템 호출을 통하여 파일 시스템과 상호작용이 일어난다.

-system call interface : ‘syscall.c’파일에서 system call 번호와 handler를 연결하고, system call을 처리한다.

-file system : ‘sysfile.c’파일에서 파일 시스템 관련 시스템 호출이 구현되어 있고, 이는 Shell script 파일과 매핑되어 백그라운드 실행을 처리한다.

```
case T_FILE:
  printf(1, "%s %d %d %d\n", fmtname(path), st.type, st.ino, st.size);
  break;
```

ls.c 커맨드 코드를 살펴보니 순서대로 파일이름, 파일 유형, inode 번호, 파일 크기를 나타내는 부분을 확인할 수 있었다.

쉘 커맨드의 처리 과정은 입력 받기 > 명령어 파싱 > 실행 파일 검색 > 프로세스 생성 및 실행 > 명령어 처리 과정 순으로 진행되며, 실행 파일 검색에서는 /bin 디렉토리 내에서 실행 파일명에 매칭된 부분을 찾고, fork() 시스템 콜을 통하여 생성된 새로운 프로세스가 execv(), execvp()

호출을 통해 파일을 실행하는 과정을 거친다. 이후 쉘은 생성된 프로세스가 종료될 때까지 대기(wait())하고, 다시 입력을 기다리는 상태로 돌아가게 된다.

3) xv6 Shell program Implementation

내장 system call을 활용한 간단한 유저 프로그램을 만들어 xv6 개발환경에 친숙해지는 시간을 가진 다음 본격적으로 OS를 발전시킬 수 있는 기능을 구현하고자 하였다. xv6에서 부팅 이후 처음으로 실행되는 유저 프로그램인 init.c를 수정하여 바로 쉘을 사용하는 것이 아닌 SSU_OS_login이 먼저 실행되고, 아이디, 비밀번호를 입력받은 후 사용자 인증을 거쳐 쉘을 사용할 수 있도록 변경하였다. 이를 통해 xv6의 Makefile 구조와 수정 방법을 익히고 User 영역과 Kernel 영역 분리를 직관적으로 확인해보고자 하였다.

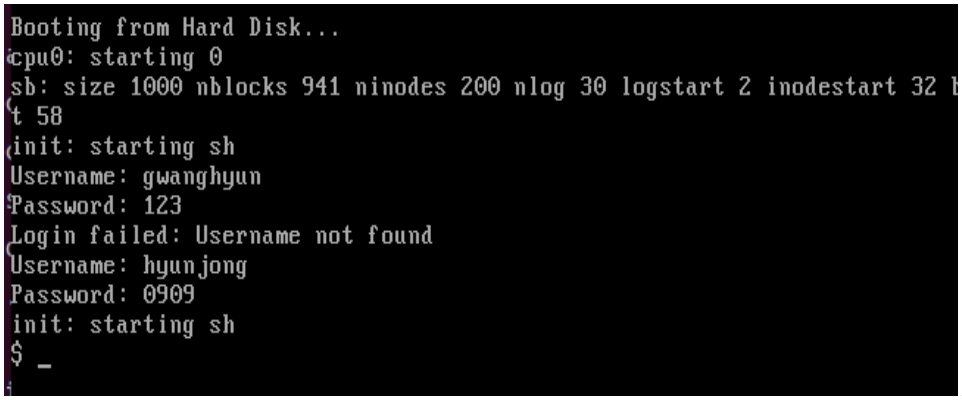
"login"은 사용자의 로그인을 처리하는 기능을 처리한다. 사용자가 로그인을 시도할 때, 시스템은 입력받은 비밀번호를 검증하고 그에 맞는 성공 혹은 실패의 출력을 내보낸다.

a) SSU_OS_login.c

이 실행 파일은 사용자로부터 입력받은 비밀번호를 검증하는 과정을 거친다. 비밀번호가 일치하지 않을 경우 "Login failed : Incorrect password" 메시지를 출력하고 "0"의 값을 반환한다. 사용자의 이름은 'list.txt'파일에 저장되어 있고, get_user_list 내 함수에서 open, read 등의 파일 시스템을 통해 userID와 pwdID 배열에 저장한다(시스템 호출 적용). check_idpw 함수는 입력받는 사용자의 이름과 비밀번호가 저장된 목록과 일치 하는지 검사하여, 만약 사용자의 이름이 시스템에 존재하지 않을 경우 "Login failed : Username not found" 메시지를 출력하고 "0"의 값을 반환한다. 사용자의 이름과 password가 매칭될 경우 "1"의 값을 반환하는 기능을 수행한다. 가장 먼저 실행되는 프로세스에 사용자 인증 기능을 추가함으로써 부팅 이후 확인을 직관적으로 구현하고자 하였다.

b) init.c

init은 시스템 부팅 후 가장 먼저 실행되는 유저 프로그램으로 다른 프로세스를 시작하고, 시스템의 기본적인 설정을 수행하는 역할을 한다. 자식 프로세스 내에서 ssu login 프로그램을 실행하도록 만들고자 exec sh 부분을 수정하였다. 기존에는 exec sh 코드가 바로 실행되었지만 login 기능을 추가하면서 login의 return값이 나온 후에 sh가 실행되도록 순서를 조정하였다.



//login 실행 화면

4) xv6 System Call Implementation(memsize,cps)

xv6에서 system call 호출 종류는 다음과 같다.

1) 인자가 없는 system 호출 EX) sysproc.c 내 구현된 uptime()
2) 문자열 및 정수 인자가 있는 system 호출 EX) sysfile.c 내 구현된 open()
3) 구조체 인자가 있는 system 호출 EX) fstat()

새로운 시스템 호출을 추가하기 위해서는 “proc.h”, “syscall.c”, “syscall.h”, “sysproc.c”, “user.h”, “usys.h”, ‘Makefile” 등에 대해서 수정이 필요하고, 다음은 실제로 xv6에서 system call을 구현할 때 추가해야할 부분들을 반영하여 “memsize()” system call을 구현한 과정이다. memsize()는 인자가 없는 system 호출로서 process의 메모리 사용량을 출력하는 기능을 수행한다.

a) “user.h” => xv6의 시스템 호출 정의
-사용자 프로그램이 새로운 시스템 호출을 사용할 수 있도록 인터페이스를 제공

```
struct stat;
struct rtcdate;

// system calls
int memsize(void);  // 새로운 시스템 호출 추가
```

b) “usys.S” => xv6의 시스템 호출 리스트
-사용자 공간에서 커널 공간으로 시스템 호출을 전달할 수 있는 어셈블리 코드를 추가

usys.S

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(memsize)   // 새로운 시스템 호출 추가
SYSCALL(fork)
```

c) “syscall.h” =>새 시스템 호출을 위해 새로운 매핑 추가.
-새로운 시스템 호출 번호를 정의하여 커널이 이를 인식할 수 있도록 변경하였다.

syscall.h > ...

```
// System call numbers
#define SYS_memsize 22   // 새로운 시스템 호출 번호 추가
#define SYS_fork    1
```

d) “syscall.c” =>시스템 호출 인수를 구문 분석하는 함수 및 실제 시스템 호출 구현에 대한 포인터
-새로운 시스템 호출을 인식하고 처리할 수 있도록 포인터 테이블과 구문 분석 함수를 수정하였다.

```
extern int sys_uptime(void);
extern int sys_memsize(void);   // 새로운 시스템 호출 추가
```

e) “sysproc.c"
프로세스 관련 시스템 호출 구현 => 시스템 호출 코드 추가

```
int
sys_memsize(void)
{
  uint size;

  size = myproc()->sz;

  return size;
}// return memory size
```

f) “proc.h” => struct proc 구조 정의

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
```

g) “memsizetest.c” => 실제 memsizetest 쉘 프로그램 구현

```
#include "types.h"
#include "stat.h"
#include "user.h"

#define SIZE 2048

int main(){
    int msize = memsize();
    printf(1, "The process is using %dB\n", msize);

    char *tmp = (char *)malloc(SIZE * sizeof(char));

    printf(1, "Allocating more memory\n");
    msize = memsize();
    printf(1, "The process is using %dB\n", msize);

    free(tmp);
    printf(1, "Freeing memory\n");
    msize = memsize();
    printf(1, "The process is using %dB\n", msize);

    exit();
}
```

h) “Makefile” => memsizetest.c
user < >system call < > kernel 확인되도록 유저 프로그램 compile 및 linking

```
UPROGS=\
    _memsizetest\
```

```
EXTRA=\
    mkfs.c ulib.c user.h cat.c ps.c echo
    ln.c ls.c mkdir.c rm.c stressfs.c us
    printf.c umalloc.c\
    README list.txt dot-bochsrc *.pl toc
    .gdbinit.tmpl gdbutil\
    memsizetest.c\
    helloworld.c\
```

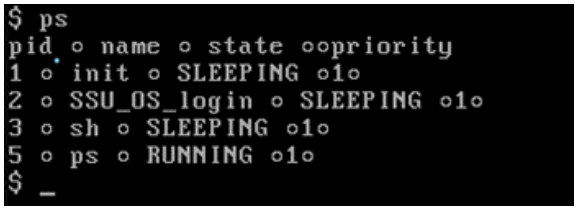
i) memsizetest system call 결과

```
init: starting SSU login
Username: minwoo
Password: 0522
init: starting sh
$ memsizetest
The process is using 24576B
Allocating more memory
The process is using 57344B
Freeing memory
The process is using 57344B
$
```

다음은 cps system call을 통해 프로세스 식별 번호(PID)를 포함하여 현재 실행 중인 프로세스에 대한 정보를 CLI로 제공하는 ps command를 동일한 방식으로 구현하였다.

```
int
cps()
{
struct proc *p;
//Enables interrupts on this processor.
sti();

//Loop over process table looking for process with pid.
acquire(&ptable.lock);
cprintf("name \t pid \t state \t priority \n");
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state == SLEEPING)
          cprintf("%s \t %d \t SLEEPING \t %d \n ", p->name,p->pid,p->priority);
        else if(p->state == RUNNING)
          cprintf("%s \t %d \t RUNNING \t %d \n ", p->name,p->pid,p->priority);
        else if(p->state == RUNNABLE)
          cprintf("%s \t %d \t RUNNABLE \t %d \n ", p->name,p->pid,p->priority);
}
release(&ptable.lock);
return 22;
}
```

```
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4  #include "fcntl.h"
5
6  int main(void){
7    cps();
8    exit();
9  }
```



출력 결과 화면

3. CPU Scheduling of xv6

1) 기존 **xv6**의 **RR scheduling**

- 스케줄링이란?

프로세스가 생성되어 실행될 때 필요한 시스템의 여러 자원을 해당 프로세스에게 할당하는 작업이다. 운영체제가 시스템 자원, 특히 CPU 시간을 효율적으로 관리하고 분배하기 위해 사용하는 메커니즘이다.

xv6는 기본적으로 선점형 Round Robin 스케줄링 알고리즘을 사용한다. 각 프로세스는 고정된 시간 동안 CPU를 할당 받고, 그 시간 동안 실행이 끝나지 않으면 대기열의 맨 뒤로 이동한다. 기본적으로 xv6에서 process state는 다음과 같이 구현 되어있다.

프로세스 상태: 생애주기에 따라 프로세스는 여러 상태 중 하나에 존재할 수 있다.

· UNUSED: 사용되지 않는 프로세스.

· EMBRYO: 생성 중인 프로세스.

· SLEEPING: 자원을 기다리면서 실행이 일시 중지된 프로세스.

· RUNNABLE: 실행 가능하지만 현재 실행 중이지 않은 프로세스.

· RUNNING: 현재 CPU에서 실행 중인 프로세스.

· ZOMBIE: 종료되었지만, 부모 프로세스가 아직 종료 상태를 수집하지 않은 프로세스.

xv6에서 각 프로세스는 struct proc 구조체로 표현되며, 이 구조체에는 프로세스 상태, 스택 포인터, 프로그램 카운터, 레지스터 상태 등 중요한 정보가 저장되어 있는 프로세스 제어 블록(PCB)의 역할을 담당한다.

```
40  struct proc {
41    uint sz;                     // Size of process memory (bytes)
42    pde_t* pgdir;                // Page table
43    char *kstack;                // Bottom of kernel stack for this process
44    enum procstate state;        // Process state
45    int pid;                     // Process ID
46    struct proc *parent;         // Parent process
47    struct trapframe *tf;        // Trap frame for current syscall
48    struct context *context;     // swtch() here to run process
49    void *chan;                  // If non-zero, sleeping on chan
50    int killed;                  // If non-zero, have been killed
51    struct file *ofile[NOFILE];  // Open files
52    struct inode *cwd;           // Current directory
53    char name[16];               // Process name (debugging)
```

// 'proc.h'에 정의된 proc 구조체

xv6에서 스케줄러는 'proc.c' 파일 내 'scheduler' 함수에서 구현된다. 함수를 보면 'RUNNABLE` 상태의 프로세스를 찾아서 CPU를 할당한다.

```
    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;
   }
   release(&ptable.lock);

  }
}
```

//'proc.c'에 정의된 scheduler 함수

또한 xv6는 Round-Robin Scheduler 구현 방식의 특징인 선점 구현을 타이머 인터럽트를 사용하여 선점을 구현한다. 'trap.c' 파일의 'trap' 함수에서 이를 처리한다.

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
```

//타이머 인터럽트 처리 코드

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

//프로세스 선점 코드

<사전작업>

컴파일 과정에서 어떤 프로세스로 돌아갈지 미리 정책을 세워서 컴파일하는 과정을 위해 Makefile을 수정해서 어떠한 스케줄링을 사용할 것인지 결정하게 한다.

- SCHED_POLICY를 설정하여 그 스케줄링 정책으로 컴파일한다.

  ex) make SCHED_POLICY=MLFQ_SCHED

```
Makefile
58  QEMU = $(shell if which qemu > /dev/null; \
70      echo "*** or have you tried setting the QEMU variable in Makefile?" 1>&2; \
71      echo "***" 1>&2; exit 1)
72  endif
73
74  SCHED_POLICY = DEFAULT
75
76  CC = $(TOOLPREFIX)gcc
77  AS = $(TOOLPREFIX)gas
78  LD = $(TOOLPREFIX)ld
79  OBJCOPY = $(TOOLPREFIX)objcopy
80  OBJDUMP = $(TOOLPREFIX)objdump
81  CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer
82  CFLAGS += $(shell $(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1 && echo -fno-stack-protector)
83  CFLAGS += -D $(SCHED_POLICY)
84  ASFLAGS = -m32 -gdwarf-2 -Wa,-divide
85  # FreeBSD ld wants ``elf_i386_fbsd''
86  LDFLAGS += -m $(shell $(LD) -V | grep elf_i386 2>/dev/null | head -n 1)
```

또한 테스트를 위한 파일들을 Makefile에 추가한다.

```
Makefile
171  UPROGS=\
190      _memsizetest\
191      _trace\
192      _ps\
193      _fcfs_test\
194      _ml_test\
195      _mlfq_test\
```

2) FCFS

FCFS 스케줄러에서는 먼저 생성된 프로세스가 먼저 스케줄링이 되도록 한다. process가 스스로 종료되거나, 200 ticks가 경과하여 종료 및 sleeping으로 전환되지 않았을때 강제 종료 시키는 경우를 제외하고는 switch-out 동작을 제한한다. 기본적으로 xv6는 스케줄러에서 타이머 인터럽트가 10ms마다 발생할 때마다 현재 프로세스를 RUNNABLE로 전환시키고 다음 프로세스를 실행시키는 Round-Robin 정책을 사용하고

있다. 반면 FCFS 스케줄러를 구현할 때는 타이머 인터럽트가 발생하더라도 RUNNABLE로 바꾸고 다음 프로세스를 실행시키는 일이 없도록 설정한다.

200ms가 되도록 스케줄링이 일어나지 않을 경우 현재 프로세스를 강제로 종료하도록 별도의 변수를 추가해야한다. 스케줄러에서 다음 실행할 프로세스를 고를 때는 상태가 RUNNABLE 한 것 중에서 가장 먼저 생성된 프로세스를 우선적으로 선정한다. 처음에는 ptable기준으로 제일 앞에 있는 프로세스부터 실행하면 된다고 간주하였다. 하지만 프로세스가 작업을 마쳤을 때 해당 인덱스로 새로운 프로세스가 들어갈 수 있으므로 ptable 인덱스 순으로 선택하면 제한사항이 발생한다. 따라서 pid가 작은 프로세스가 먼저 생성되었음을 의미하므로 pid를 이용해 가장 먼저 생성된 프로세스를 선별한다.

다음은 실제 구현 과정 및 결과이다.

a. 타이머 인터럽트가 발생하더라도 다음 프로세스를 실행하지 않게 하되 스케줄러가 실행된지 200tick이 넘으면 강제종료 시켜야 한다. 스케줄러가 마지막으로 실행된 지 얼마나 시간이 경과했는지를 체크하기 위한 별도의 변수 int ticks_FCFS를 다음과 같이 trap.c, proc.c에 정의한다.스케줄러가 실행할 때마다 이 값은 0으로 초기화되어야 한다.

trap.c

```c
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.h"

// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
extern uint vectors[];  // in vectors.S: array of 256 entry pointers
struct spinlock tickslock;
uint ticks;

int ticks_FCFS = 0;
```

proc.c

```c
int nextpid = 1;
extern void forkret(void);
extern void trapret(void);

static void wakeup1(void *chan);

extern int ticks_FCFS;
```

```
C proc.c
428   //      via swtch back to the scheduler.
429   void
430   scheduler(void){
431     struct proc *p;
432     struct cpu *c = mycpu();
433     c->proc = 0;
434
435     ticks_FCFS=0;
```

b. 기존 xv6에서는 타이머 인터럽트가 발생할 때마다 yield로 다른 프로세스에게 cpu를 양보한다. 이 부분을 FCFS스케줄러로 xv6를 실행할 때는 yield를 하는 대신 ticks_FCFS값을 1 증가시킨다. scheduler함수가 실행될 때마다 초기화되는 ticks_FCFS가 200을 넘었다는 것은 프로세스가 스케줄링 된 이후 200 ticks가 지날 때까지 종료되거나 SLEEPING 상태로 전환되지 않았다는 것이므로 현재 프로세스를 kill시키고 다른 프로세스에게 cpu를 넘긴다.

```
C trap.c
46  {
113   if(myproc() && myproc()->state == RUNNING &&
114      tf->trapno == T_IRQ0+IRQ_TIMER) {
115
116   #ifdef FCFS_SCHED
117     ticks_FCFS++;
118     if(ticks_FCFS>200) { // over 200 ticks
119       myproc()->killed = 1;
120       yield();
121     }
```

c. 다음으로 proc.c에 있는 scheduler함수를 변경한다. 먼저 FCFS 스케줄러로 실행할 때는 가장 작은 pid 값의 프로세스를 저장하도록 struct proc *minPid 변수를 선언한다. 초기값은 NULL값을 가진다.

```
429   void
430   scheduler(void){
431     struct proc *p;
432     struct cpu *c = mycpu();
433
434   #ifdef FCFS_SCHED
435     struct proc *minPid;
```

d. 다음으로 다른 프로세스를 선택하는 과정을 바꿔줘야 한다. 기존 xv6에서는 ptable을 돌면서 상태가 RUNNABLE인 프로세스를 발견하는 순간 그 프로세스를 대상으로 cpu를 할당한다. 이 과정을 FCFS 스케줄러로 실행하는 경우에는 ptable을 모두 확인하고, pid가 가장 작은 프로세스를 minPid에 넣어서 이후 해당 프로세스에게 cpu를 할당한다. minPid를 정하고 난 뒤 cpu를 주는 부분은 기존 xv6 코드에서 p를 minPid로 수정한다.

```
459   #ifdef FCFS_SCHED
460     minPid = 0;
461     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
462       if(p->state != RUNNABLE)
463         continue;
464
465       if(minPid == 0) { // inital runnable
466         minPid=p;
467       }
468       else { // update
469         if(minPid->pid > p->pid)
470           minPid = p;
471       }
472     }
473
474     if(minPid !=0) { // execute minPid (change p to minPid)
475       c->proc = minPid;
476       switchuvm(minPid);
477       minPid->state = RUNNING;
478       swtch(&(c->scheduler), minPid->context);
479       switchkvm();
480       c->proc = 0;
481     }
```

proc.c

```
596   #else
597     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
598       if(p->state != RUNNABLE)
599         continue;
600
601       // Switch to chosen process.  It is the process's job
602       // to release ptable.lock and then reacquire it
603       // before jumping back to us.
604       c->proc = p;
605       switchuvm(p);
606       p->state = RUNNING;
607
608       swtch(&(c->scheduler), p->context);
609       switchkvm();
610
611       // Process is done running for now.
612       // It should have changed its p->state before coming back.
613       c->proc = 0;
614     }
```

추가적으로 yield를 유저프로그램에서도 사용할 수 있도록 시스템콜을 만들어줘야 테스트파일을 정상적으로 실행할 수 있는데, 이 부분은 트러블슈팅 부분에서 구현하였다.

e. 파일을 모두 위와 같이 변경한 후 make SCHED_POLICY=FCFS_SCHED -> make qemu 순으로 컴파일을 진행하고 xv6에서 fcfs_test 파일을 실행한 결과 아래와 같이 결과가 정상적으로 나오는

것을 확인할 수 있다. 일반적인 경우, yield를 하는 경우 모두 pid순으로 잘 출력되는 것을 확인할 수 있다. 다만 sleep을 하는 경우에는 다소 순서가 뒤죽박죽인데, 이는 실행중인 프로세스가 sleeping 상태로 전환된 경우 다음으로 pid가 작은 프로세스가 스케줄링이 되기 때문이다. sleep을 하다 깨어나면 그 프로세스가 다시 최소 pid 프로세스가 되므로 도중에 더 앞의 번호의 pid가 적히는 것이다.

```
init: starting sh
$ fcfs_test
FCFS test start

자식 프로세스를 5번 실행

 process 5  process 5  process 5  process 5  process 5
 process 6  process 6  process 6  process 6  process 6
 process 7  process 7  process 7  process 7  process 7
 process 8  process 8  process 8  process 8  process 8
 process 9  process 9  process 9  process 9  process 9
```

```
루프돌며 자신 PID 출력하고 CPU 양보

 process 10  process 10  process 10  process 10  process 10
 process 11  process 11  process 11  process 11  process 11
 process 12  process 12  process 12  process 12  process 12
 process 13  process 13  process 13  process 13  process 13
 process 14  process 14  process 14  process 14  process 14
```

```
자신 PID 출력하고 1초 대기
 process 15
 process 16
 process 17
 process 15
 process 16
 process 15
 process 16
 process 17
 process 18
 process 15
 process 16
 process 17
 process 15
 process 16
 process 17
 process 18
 process 17
 process 18
 process 19
 process 18
 process 19
 process 18
 process 19
 process 19
 process 19
$
```

<Trouble Shooting>

테스트 파일 fcfs_test와 함께 make를 할 때 오류가 발생하였다. yield함수를 찾을 수 없어 오류가 발생하였다. 원인을 분석한 결과 yield함수가 xv6에 존재하는 것은 맞으나 커널안에서 사용할 수 있게만 정의되어 있고 유저 프로그램에서 사용할 수 있게 하지는 않았기 때문이다. 따라서 yield 시스템 콜을 유저프로그램에서 사용할 수 있도록 다음과 같은 조치를 진행하였다. proc.c에서 yield의 wrapper function을 정의하였다. yield는 리턴하지 않는 함수이지만 다른 시스템콜과 반환형을 맞추기 위해 0을 리턴하도록 하였다. 추가로 proc.h에서도 해당 함수를 선언하였다.

proc.c

```
630 {
644     mycpu()->intena = intena;
645   }
646
647   int
648   sys_yield(void)
649   {
650     yield();
651     return 0;
652   }
```

proc.h

```
34
35   enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
36
37   int sys_yield(void);
```

syscall.h, syscall.c에서 yield 시스템콜을 다음과 같이 번호를 부여하며 추가하였다.

syscall.h

```
13   #define SYS_sbrk   12
14   #define SYS_sleep  13
15   #define SYS_uptime 14
16   #define SYS_open   15
17   #define SYS_write  16
18   #define SYS_mknod  17
19   #define SYS_unlink 18
20   #define SYS_link   19
21   #define SYS_mkdir  20
22   #define SYS_close  21
23   #define SYS_memsize 22   // 새로운 시스템 호출 번호 추가
24   #define SYS_trace 23     // 추가
25   #define SYS_ps 24
26   #define SYS_yield 25
```

```c
// syscall.c
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_memsize(void);   // 새로운 시스템 호출 추가
extern int sys_trace(void); // 추가
extern int sys_ps(void);
extern int sys_yield(void);
```

```c
// syscall.c
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_memsize]   sys_memsize,   // 새로운 시스템 호출 추가
[SYS_trace]   sys_trace, // 추가
[SYS_ps]   sys_ps, // 추가
[SYS_yield]   sys_yield,
```

마지막으로 구현한 시스템콜을 유저 프로그램에서 사용할 수 있도록 user.h와 usys.S에 다음의 내용을 추가하였다.

```c
// user.h
int uptime(void);
int memsize(void);  // 새로운 시스템 호출 추가
int trace(int);   // 새로운 시스템 호출 추가
int ps(void);       // ps 커맨드
void yield(void);
```

```
// usys.S
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(memsize)   // 새로운 시스템 호출 추가
SYSCALL(trace)     // 새로운 시스템 호출 추가
SYSCALL(ps)
SYSCALL(yield)
```

위와 같은 방법으로 유저 프로그램에서도 yield를 사용할 수 있도록 시스템콜을 구현하였다.

3) Multi-level Queue(FCFS, RR)

Multilevel Queue에서는 Round robin 방식과 FCFS 방식을 각각 이용하는 2개의 큐로 이루어지고, pid가 짝수인 프로세스는 RR으로, pid가 홀수인 프로세스는 FCFS로 스케줄링한다. 또한 RR큐가 FCFS큐보다 높은 우선수위를 가지도록 구성하였다. 따라서 스케줄러에서 RR큐와 FCFS큐 두 개로 나누고, RR큐를 먼저 봐서 RUNNABLE한 프로세스를 실행한 후, 만약 RR큐를 모두 보아도 runnable 프로세스가 없을 경우에 그제서야

FCFS큐를 보도록 하면 된다. RR큐에서는 원래 xv6가 하던 방식 그대로 가져오면 round robin으로 스케줄링 될 것이며, FCFS큐에서는 위에서 했던대로 pid가 가장 작은 것을 선택해서 스케줄링을 하면 된다.

다음은 실제 구현 과정 및 결과이다.

a. proc.h에 있는 proc 구조체에 int queueLevel 변수를 추가한다. 이 값은 해당 프로세스가 어느 큐에 들어가 있는지를 나타내며, queueLevel=0이면 RR큐에, queueLevel=1이면 FCFS큐에 들어가 있음을 의미한다.

proc.h

```
40    struct proc {
53      char name[16];               // Process name (debugging)
54      int tracemask;     // add field 05_17 tracemask
55      int priority;      // add 05_19 ps commend
56    #ifdef MULTILEVEL_SCHED
57      int queueLevel;
```

b. proc.c에 있는 allocproc함수에서 pid에 따라 서로 다른 큐에 집어넣도록 다음 코드를 추가한다. pid가 짝수이면 RR큐에, pid가 홀수이면 FCFS큐에 들어가게 된다.

proc.c

```
96    found:
100     #ifdef MULTILEVEL_SCHED
101     if((p->pid)%2 == 0)
102       p->queueLevel = 0; // Round Robin queue
103     else
104       p->queueLevel = 1; // FCFS queue
```

c. RR큐에서는 타이머 인터럽트가 올 때마다 다른 프로세스를 실행해야 하나, FCFS큐에서는 pid가 작은 프로세스가 끝까지 실행해야 한다. 따라서 다음과 같이 타이머 인터럽트가 올 때 RR큐에서만 yield를 하도록 한다.

```
C trap.c
 46   {
113     if(myproc() && myproc()->state == RUNNING &&
116   #ifdef FCFS_SCHED
117     ticks_FCFS++;
118     if(ticks_FCFS>200) { // over 200 ticks
119       myproc()->killed = 1;
120       yield();
121     }
122   #elif MULTILEVEL_SCHED
123     if(myproc()->queueLevel == 0) { // if Round Robin queue
124       yield();
125     }
```

d. scheduler 함수에서 targetP 포인터변수를 추가한다. minPid 때와 유사하게 이 변수는 RR큐에 runnable한게 있으면 RR큐의 프로세스를 가리키고, 없을 경우 FCFS큐에서 가장 pid가 작은 프로세스를 가리키게 된다.

```
C proc.c
429   void
430   scheduler(void){
431     struct proc *p;
432     struct cpu *c = mycpu();
433
434   #ifdef FCFS_SCHED
435     struct proc *minPid;
436   #elif MULTILEVEL_SCHED
437     struct proc *targetP;
```

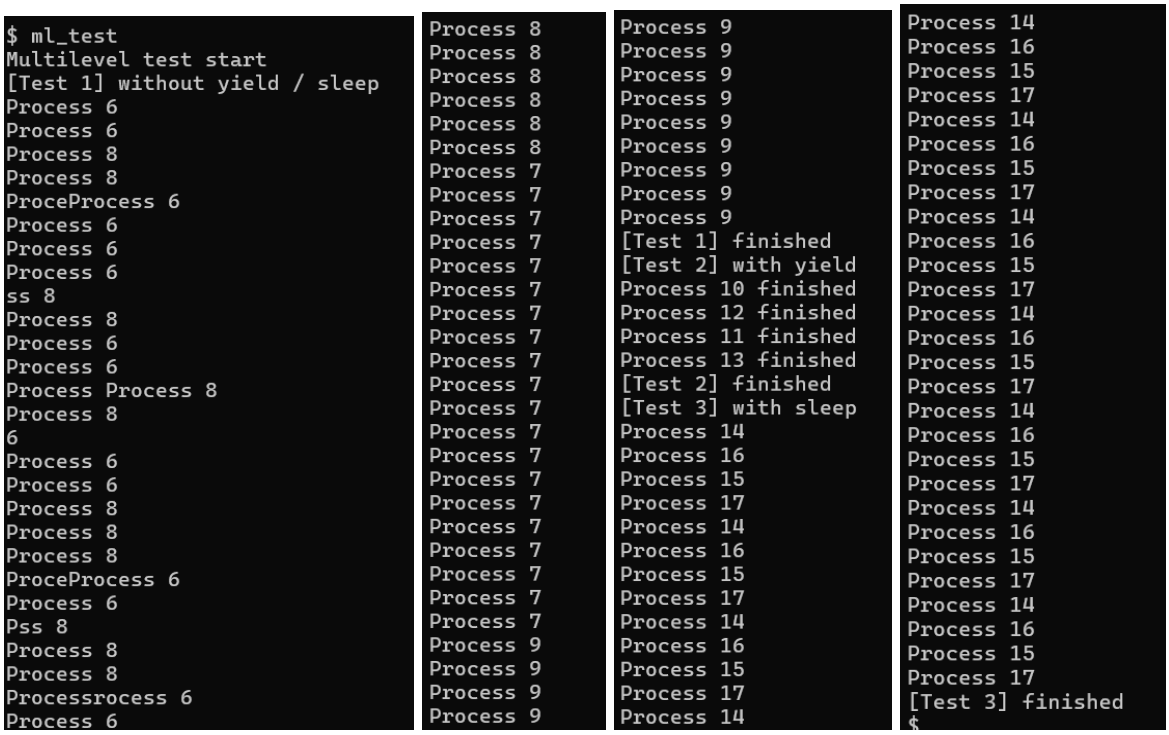
e. 이후 다음 프로세스를 고르는 부분을 다음과 같이 추가한다. RR큐를 먼저 체크해서 runnable한게 있으면 그것을 우선으로 targetP로 설정하고, runnable한 게 없으면 FCFS큐를 모두 봐서 pid가 가장 작은 프로세스를 targetP로 설정하게 된다. 이후 해당 선택한 targetP 프로세스에게 CPU를 주게 된다.

proc.c

```
482   #elif MULTILEVEL_SCHED
483     targetP = 0;
484     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
485       if(p->state != RUNNABLE || p->queueLevel != 0) // check RR queue
486         continue;
487       targetP = p;
488
489       c->proc = p;
490       switchuvm(p);
491       p->state = RUNNING;
492
493       swtch(&(c->scheduler), p->context);
494       switchkvm();
495
496       c->proc = 0;
497     }
498
499     if(targetP == 0) { // check FCFS queue if no RUNNABLE in RR queue
500       for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
501         if(p->state != RUNNABLE || p->queueLevel != 1)
502           continue;
503
504         if(targetP == 0) { // initial runnable
505           targetP=p;
506         }
```

```
507         else { // update minPid
508           if(targetP->pid > p->pid)
509             targetP = p;
510         }
511       }
```

f. 파일을 모두 위와 같이 변경한 후 make SCHED_POLICY=MULTILEVEL_SCHED -> make qemu 순으로 컴파일을 진행하고 xv6에서 ml_test 파일을 실행한 결과 아래와 같이 결과가 정상적으로 나오는 것을 확인할 수 있다. 우선 pid가 짝수인걸 먼저 실행하고, pid가 짝수인 runnable 프로세스가 없으면 pid가 홀수인 프로세스가 pid가 작은 것부터 실행된다. RR큐에서는 10ms마다 cpu를 잡는 프로세스가 바뀌므로 pid가 짝수인 것 끼리는 순서가 다음과 같이 섞일 수 있게 된다. -> printf 함수가 버퍼링되어 여러 프로세스에서 동시에 출력할 때 출력이 섞이는 경우 발생한다.



<Trouble Shooting>

처음에 MULTILEVEL 스케줄러에서 scheduler함수를 작성할 때 다음 아래 사진의 코드를 작성하였다. RR큐에 있는 프로세스에 대하여 ptable의 처음부터 끝까지 확인하면서 runnable인게 있으면 바로 targetP에 저장하고 반복문을 탈출한 뒤 이를 대상으로 CPU를 주는 방식으로 작성하였다. 다만 이 경우 RR큐에서 scheduler가 일어날 때마다 ptable의 처음부터 확인을 하는 구조가 되어 만약 ptable 처음에 있는 프로세스의 경우 yield가 일어난다 하더라도 첫 프로세스가 runnable이기 때문에 다시 cpu를 잡는 상황이 생기게 되어 아래 실행결과 같이 RR큐에서 10ms마다 돌아가지 않고 처음 cpu를 잡은 프로세스가 끝까지 실행하는 상황이 일어나게 된다. 이러한 문제를 해결하기 위해 위에서와 같이 코드를 수정하였고 이후 Round robin 방식이 정상적으로 잘 된 것을 확인할 수 있었다.

proc.c

```c
#elif MULTILEVEL_SCHED
  targetP = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->state != RUNNABLE || p->queueLevel != 0) // check RR queue
      continue;
    targetP = p;
  }

  if(targetP == 0) { // check FCFS queue if no RUNNABLE in RR queue
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if(p->state != RUNNABLE || p->queueLevel != 1)
        continue;

      if(targetP == 0) { // initial runnable
        targetP=p;
      }
      else { // update minPid
        if(targetP->pid > p->pid)
          targetP = p;
      }
    }
```

```c

    if(targetP != 0) { // execute targetP (change p to targetP)
      c->proc = targetP;
      switchuvm(targetP);
      targetP->state = RUNNING;
      swtch(&(c->scheduler), targetP->context);
      switchkvm();
      c->proc = 0;
    }
  }
```

```
$ ml_test
Multilevel test start
[Test 1] without yield / sleep
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
Process 6
```

```
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 8
Process 5
Process 5
Process 5
Process 5
Process 5
Process 5
```

```
Process 5
Process 5
Process 5
Process 5
Process 5
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
Process 7
```

```
[Test 1] finished
[Test 2] with yield
Process 10 finished
Process 12 finished
Process 9 finished
Process 11 finished
[Test 2] finished
[Test 3] with sleep
Process 14
Process 16
Process 13
Process 15
Process 14
Process 16
Process 13
Process 15
Process 14
Process 16
Process 13
Process 15
Process 14
Process 16
Process 13
Process 15
Process 14
Process 16
Process 13
Process 15
Process 14
Process 16
```

```
Process 16
Process 13
Process 15
Process 14
Process 16
Process 13
Process 15
Process 14
Process 16
Process 13
Process 15
[Test 3] finished
```

4) MLFQ

이번에 구현하는 MLFQ에서는 기본 RR 정책을 따르되 조건에 따라 L0큐와 L1큐 사이를 이동할 수 있어야 한다. Multilevel 큐를 구현했을 때와 마찬가지로 queueLevel을 두도록 하되, monopolize를 통해 CPU를 독점하는 경우 최우선적으로 권한을 가지는 별도의 큐를 만들어 그 것으로 이동하는 방법을 취한다(queueLevel을 100/101으로 설정한 큐로 이동시키고, 스케줄러에서는 이 큐를 가장 먼저 체크하여 있을 경우 이를 최우선으로 처리한다). 또한 타이머 인터럽트가 발생할 때마다 해당 프로세스에게 주어진 time quantum이 감소하여 모두 사용한 경우 L1큐로 내려가거나 우선순위가 내려가게 해야 하며, 200 ticks가 지날 때마다 L0큐/priority0으로 초기화를 해줘야 한다.

다음은 실제 구현 과정 및 결과이다.

a. Multilevel queue 때와 유사하게 proc.h에 정의된 proc구조체에 int queueLevel2와 int priority2 변수를 추가한다. 이들은 각각 위치한 큐의 종류와 우선순위를 나타낸다. (MULTILEVEL일 때와 구분하기 위해 뒤에 숫자 2를 붙여주었다) 또한 proc.c의 allocproc함수에서 프로세스가 만들어질 때 이들은 모두 0으로 설정된다. (L0큐, 우선순위 0)

```
C proc.h
40   struct proc {
53     char name[16];               // Process name (debugging)
54     int tracemask;    // add field 05_17 tracemask
55     int priority;     // add 05_19 ps commend
56   #ifdef MULTILEVEL_SCHED
57     int queueLevel;
58   #elif MLFQ_SCHED
59     int queueLevel2;
60     int priority2;
61   #endif
62   };
```

```
C proc.c
96   found:
103    else
105  #elif MLFQ_SCHED
106    p->queueLevel2 = 0;
107    p->priority2 = 0;
108  #endif
109    release(&ptable.lock);
```

b. proc.c의 scheduler함수에서 cpu를 줄 프로세스를 지정하는 targetP2 포인터변수를 추가한다. 이후 다음으로 cpu를 줄 프로세스를 선정할 때 1. monopolize 상태의 프로세스 -> 2. L0큐에 있는 runnable 프로세스 -> 3. L1큐에 있는 runnable 프로세스 순으로 찾게 된다. (L1큐안에서 프로세스를 선정할 때는 우선순위가 큰 경우 -> pid가 작은 경우 순으로 프로세스 우선순위를 갖는다)

proc.c

```
429  void
430  scheduler(void)
431  {
432    struct proc *p;
433    struct cpu *c = mycpu();
434
435  #ifdef FCFS_SCHED
436    struct proc *minPid;
437  #elif MULTILEVEL_SCHED
438    struct proc *targetP;
439  #elif MLFQ_SCHED
440    struct proc *targetP2;
441  #endif
```

proc.c

```
524  #elif MLFQ_SCHED
538    targetP2 = 0;
539    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) { // first check monopolized process
540      if(p->state != RUNNABLE || p->queueLevel2 < 100)
541        continue;
542    }
543
544    if(targetP2 != 0) {
545      c->proc = targetP2;
546      switchuvm(targetP2);
547      targetP2->state = RUNNING;
548      swtch(&(c->scheduler), targetP2->context);
549      switchkvm();
550      c->proc = 0;
551    }
552
553    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
554      if(p->state != RUNNABLE || p->queueLevel2 != 0) // check L0
555        continue;
556      targetP2 = p;
557
558      c->proc = p;
559      switchuvm(p);
560      p->state = RUNNING;
561
562      swtch(&(c->scheduler), p->context);
563      switchkvm();
564
565      c->proc = 0;
566    }
```

```
524  #elif MLFQ_SCHED
568    if(targetP2 == 0) { // check L1
569      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
570        if(p->state != RUNNABLE || p->queueLevel2 != 1)
571          continue;
572
573        if(targetP2 == 0) { // initial runnable
574          targetP2 = p;
575        }
576        else { // update targetP2
577          if(targetP2->priority2 < p->priority2) {
578            targetP2 = p;
579          }
580          else if(targetP2->priority2 == p->priority2 && targetP2->pid > p->pid) {
581            targetP2 = p;
582          }
583        }
584      }
585
586      if(targetP2 != 0) {
587        c->proc = targetP2;
588        switchuvm(targetP2);
589        targetP2->state = RUNNING;
590        swtch(&(c->scheduler), targetP2->context);
591        switchkvm();
592        c->proc = 0;
593      }
594    }
```

c. 또한 프로세스가 실행될 때 특정 tick이 경과할 때마다 큐이동/priority boost의 조치를 취해줘야 하므로 이 부분도 구현해야 한다. 먼저 trap.c에서 ticks_L0, ticks_L1, ticks_boost 변수를 추가한다. ticks_L0은 프로세스가 L0큐에서 돌아갈 때 1씩 감소하며 0이 되면 L1으로 떨어지며 yield를 호출하고 다시 4로 초기화 된다. ticks_L1은 프로세스가 L1큐에서 돌아갈 때 1씩 감소하며 0이 되면 우선순위가 1 감소하며 yield를 호출하고 다시 8로 초기화된다. ticks_boost는 어느 큐에 있든 상관없이 1씩 감소하며 0이 되면 모든 프로세스를 L0큐로 이동시키고 priority도 0으로 초기화한다 (다만 이 때 monopolize 상태의 프로세스는 건드리지 않으며 원래 큐를 L1에서 L0으로 바꿔주기만 한다) ticks가 1씩 감소하는 것은 trap.c에서 타이머 인터럽트가 왔을 때 처리해주며 ticks가 0이하일 때 처리를 하는 것은 proc.c의 스케줄러에서 해준다. 이로써 기본적인 MLFQ의 구현은 완료되었다.

C trap.c

```
19
20   #ifdef MLFQ_SCHED
21   int ticks_L0 = 4;
22   int ticks_L1 = 8;
23   int ticks_boost = 200;
24   #endif
```

trap.c

```
126   #elif MLFQ_SCHED
127     ticks_boost--;
128     if(myproc()->queueLevel2 == 0) {
129       ticks_L0--;
130       if(ticks_L0 <= 0) {
131         myproc()->queueLevel2 = 1; // go L1 queue
132         yield();
133       }
134     }
135     else if(myproc()->queueLevel2 == 1) {
136       ticks_L1--;
137       if(ticks_L1 <= 0) {
138         if(myproc()->priority2 > 0)
139           myproc()->priority2--; // priority2 down
140         yield();
141       }
142     }
143
144   #else
145     yield();
146   #endif
147     }
```

proc.c

```
24
25  #ifdef MLFQ_SCHED
26  extern int ticks_L0;
27  extern int ticks_L1;
28  extern int ticks_boost;
29  #endif
```

```
444 #ifdef MLFQ_SCHED
445   ticks_L0 = 4;
446   ticks_L1 = 8;
447 #endif
```

proc.c

```
524   #elif MLFQ_SCHED
525     if(ticks_boost <= 0) { // priority2 boost before select process
526       ticks_boost = 200;
527
528       for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
529         if(p->queueLevel2 == 101) // maintain monopolize status but change L0 after remove monopolize status
530           p->queueLevel2 = 100;
531         if(p->queueLevel2 == 1) // L1 to L0
532           p->queueLevel2 = 0;
533         p->priority2 = 0;
534       }
535     }
```

d. 마지막으로 지정한 system call을 구현하고 유저프로그램에서도 사용할 수 있게 해야 한다. getlev함수는 현재 어디 큐에 있는지 큐의 레벨을 반환한다. %100을 한 것은 기본적으로 monopolize 상태인 프로세스를 queueLevel에 100을 더해서 표시하기 때문이며, 이 경우 원래 큐레벨을 표시하기 위해 100으로 나눈 나머지를 반환한 것이다.

proc.c

```
288  int
289  getlev(void)
290  {
291  #ifdef MLFQ_SCHED
292    return (myproc()->queueLevel2)%100;
293  #else
294    return 0;
295  #endif
296  }
```

setpriority함수는 pid와 priority값을 받아 지정한 프로세스의 priority값을 변경해준다. 만약 priority값이 유효하지 않으면 -2를 리턴하고, ptable에서 해당 pid를 찾을 수 없으면 -1을 리턴한다. 정상적으로 변경했을 경우 0을 리턴한다. 이 때 ptable을 확인하기 전 lock을 획득해서 race condition을 막는다.

proc.c

```
303
304  int
305  setpriority(int pid, int priority)
306  {
307  #ifdef MLFQ_SCHED
308    struct proc *p;
309    if(priority < 0 || priority > 10)
310      return -2;
311
312    acquire(&ptable.lock);
313    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
314      if(pid == p->pid) {
315        p->priority2 = priority;
316        release(&ptable.lock);
317        return 0;
318      }
319    }
320
321    release(&ptable.lock);
322    return -1;
323  #else
324    return 0;
325  #endif
326  }
```

monopolize함수는 암호(2024)를 입력받아 암호가 맞다면 스케줄러의 최우선순위를 가지게 된다. queueLevel2의 값을 100을 더하는 것으로 최우선 순위의 큐에 넣어주는 효과를 가진다. 만약 이미 queueLevel2의 값이 100 이상이면 이미 monopolize상태인 것이므로 L0큐/priority 0으로 초기화해준다. 만약 입력받은 암호가 틀렸다면 현재 프로세스를 kill하고 yield로 다른 프로세스를 부르도록 한다.

proc.c

```
344  void
345  monopolize(int password)
346  {
347  #ifdef MLFQ_SCHED
348    if(password != 2024) {
349      myproc()->killed = 1;
350      yield();
351    }
352
353    if(myproc()->queueLevel2 >= 100) { // already monopolize state
354      myproc()->queueLevel2 = 0; // go L0 queue
355      myproc()->priority2 = 0;
356    }
357    else {
358      myproc()->queueLevel2 += 100;
359    }
360  #else
361    // nothing to do
362  #endif
363  }
```

각각의 함수는 MLFQ_SCHED가 아닌 경우 단순히 0을 리턴하거나 아무것도 안 하는 함수가 된다.

e. 또한 정의한 함수를 systemcall로 만들고 유저프로그램에서 이용할 수 있도록 코드를 추가해야 한다. 다음과 같이 wrapper function을 추가하고 defs.h, syscall.h, syscall.c, user.h, usys.S에 다음의 코드를 추가해 주어 system call로 만들고 유저프로그램에서도 이용할 수 있게 한다.

defs.h

```
122  void            yield(void);
123  int             ps(void);
124  int             getlev(void);
125  int             setpriority(int, int);
126  void            monopolize(int);
```

proc.c

```
298  int
299  sys_getlev(void)
300  {
301    return getlev();
302  }
```

proc.c

```c
int
sys_setpriority(void)
{
  int *n1;
  int *n2;
  int a = 0, b = 0;
  n1 = &a;
  n2 = &b;

  if(argint(0, n1) < 0)
    return -1;
  if(argint(1, n2) < 0)
    return -1;
  return setpriority(*n1, *n2);
}
```

proc.c

```c
int
sys_monopolize(void)
{
  int *n;
  int a = 0;
  n = &a;
  if(argint(0, n) < 0)
    return -1;
  monopolize(*n);
  return 0;
}
```

syscall.h

```c
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_memsize 22   // 새로운 시스템 호출 번호 추가
#define SYS_trace 23     // 추가
#define SYS_ps 24
#define SYS_yield 25
#define SYS_getlev 26
#define SYS_setpriority 27
#define SYS_monopolize 28
```

syscall.c

```
106  extern int sys_memsize(void);   // 새로운 시스템 호출 추가
107  extern int sys_trace(void); // 추가
108  extern int sys_ps(void);
109  extern int sys_yield(void);
110  extern int sys_getlev(void);
111  extern int sys_setpriority(void);
112  extern int sys_monopolize(void);
```

syscall.c

```
138  [SYS_ps]    sys_ps, // 추가
139  [SYS_yield]    sys_yield,
140  [SYS_getlev]    sys_getlev,
141  [SYS_setpriority] sys_setpriority,
142  [SYS_monopolize] sys_monopolize,
143  };
```

user.h

```
26  int memsize(void);  // 새로운 시스템 호출 추가
27  int trace(int);   // 새로운 시스템 호출 추가
28  int ps(void);       // ps 커맨드
29  void yield(void);
30  int getlev(void);
31  int setpriority(int, int);
32  void monopolize(int);
```

usys.S

```
30  SYSCALL(sleep)
31  SYSCALL(uptime)
32  SYSCALL(memsize)   // 새로운 시스템 호출 추가
33  SYSCALL(trace)     // 새로운 시스템 호출 추가
34  SYSCALL(ps)
35  SYSCALL(yield)
36  SYSCALL(getlev)
37  SYSCALL(setpriority)
38  SYSCALL(monopolize)
```

f. 앞에서와 같은 방법으로 make SCHED_POLICY=MLFQ_SCHED -> make qemu 순으로 컴파일을 진행하고 mlfq_test 파일을 실행한 결과 다음과 같이 결과가나온 것을 확인할 수 있다. 각 프로세스가 test1에서는 20000번, test2에서는 50000번, test3에서는 10000번, test4에서는 25000번 반복문을 돌며 L0큐에 있으면 cnt[0], L1큐에 있으면 cnt[1]값을 올리며 이 둘의 값을 합치면 정상적으로 제대로 count가 된 것을 확인할 수 있다. test4의 경우 마지막 프로세스가 monopolize 되기 때문에 L0큐에서 25000번의 연산을 모두 하였고, 끝난 것도 가장 먼저 끝난 것을 확인할 수 있다.

```
init: starting sh
$ mlfq_test
MLFQ test start

Focused priority
process 9: L0=408, L1=19592
process 8: L0=253, L1=19747
process 7: L0=609, L1=19391
process 6: L0=675, L1=19325
process 5: L0=507, L1=19493

Without priority manipulation
process 10: L0=1396, L1=48604
process 11: L0=1215, L1=48785
process 12: L0=1611, L1=48389
process 13: L0=2154, L1=47846
process 14: L0=1408, L1=48592

With yield
process 15: L0=50, L1=9950
process 16: L0=429, L1=9571
process 17: L0=302, L1=9698
process 18: L0=7, L1=9993
process 19: L0=284, L1=9716

Monopolize
process 24: L0=25000, L1=0
process 20: L0=924, L1=24076
process 21: L0=1167, L1=23833
process 22: L0=880, L1=24120
process 23: L0=1164, L1=23836
$
```

4. Memory Allocation

1) 기존 xv6 memory allocation

xv6의 메모리 할당 기법은 4KB의 고정 크기 할당을 통한 페이지로 구성되며, 연결리스트를 사용하여 사용가능한 페이지의 목록을 유지하고 있다. 유저 프로세스 영역 이외에 커널 영역도 메모리를 자체적인 힙 영역을 통해 관리하며, kalloc() 및 kfree()함수를 통해 메모리 할당과 해제를 관리하고 있다. 커널의 적재 위치 방식은 0x80000000 번지부터 시작되어 메모리의 상위 주소 부분에 위치하고 있으며 내부적으로 페이지 할당자를 사용하여 커널의 메모리 요구를 충족하는 방식이다.

간단한 형태의 가상 메모리 방식을 사용하고 있는데 프로세스를 논리 메모리 형태로 바라본 후 자신만의 주소 공간에서 실행되는 것처럼 작동하게 하기 위함이다. 현대 컴퓨터의 메모리 관리 기법과는 다르게 공유 메모리 영역을 제공하지 않고 기본적으로 다른 프로세스의 메모리 영역에는 접근이 불가하다. 프로세스의 가상 주소를 실제 물리 주소로 매핑하기 위해 페이지 테이블을 사용하고 있는데 계층적 페이지 테이블 구조로, 상위 비트로 디렉터리 엔트리를 찾고, 중간 비트를 사용하여 특정 페이지의 물리 주소를 결정하여 사용한다.

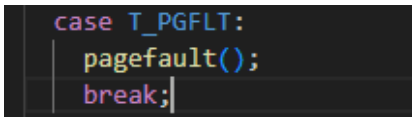
프로세스가 생성되면서 페이지 디렉터리가 생성되면, 필요한 페이지 테이블이 초기화되면서 커널 영역과 유저 영역을 구분하여 매핑된다. xv6는 CPU의 페이지 테이블(디렉터리) 베이스 레지스터(PTBR//CR3)를 사용하여 context switching을 진행하고, 프로세스가 실행되는 동안 새로운 페이지를 할당하거나 기존 페이지가 반환될 때 페이지 테이블을 동적 업데이트하는 방식의 메모리 관리 기법을 사용하여 설계되었다.

2) page fault

xv6에 구현되어 있지 않은 기능 page fault를 구현하기 위해 user 영역에 할당된 메모리 공간과 실제 물리 메모리가 가상 메모리로 맵핑되는 과정을 확인하였다.

2

xv6에서 page fault는 interrupt 발생 > fault address 확인 > page table 접근 > 물리 메모리가 할당 가능할 경우 allocate 이후 재실행 // 할당 불가할 경우 기존의 프로세스를 종료시키거나 panic 을 통한 HW down flow로 구현할 수 있다.



trap.c 파일 내에서 페이지 폴트 시 함수 호출하도록 트랩 연결

```
//PAGEFAULT
void pagefault(void)
{
  char* mem;
  pte_t* pte;
  uint addr = rcr2();  // 페이지 폴트 발생 주소 확인
  struct proc *curproc = myproc();

  // 폴트 주소 범위 유효성 확인
  if (addr >= KERNBASE || addr < curproc->sz || addr < (curproc->tf->esp - PGSIZE * 4)) {
    cprintf("[Pagefault] Invalid access at address: 0x%x\n", addr);
    curproc->killed = 1;
    return;
  }

  // 페이지 테이블 항목 로드
  pte = walkpgdir(curproc->pgdir, (char*)PGROUNDDOWN(addr), 0);

  if (pte == 0) {
    cprintf("pagefault: walkpgdir failed\n");
    curproc->killed = 1;
    return;
  }

  //물리 페이지 할당
  mem = kalloc();
  if (mem == 0) {
    cprintf("pagefault: kalloc failed\n");
    curproc->killed = 1;
    return;
  }
  memset(mem, 0, PGSIZE);

  // 새로운 페이지를 페이지 테이블에 매핑
  if (mappages(curproc->pgdir, (char*)PGROUNDDOWN(addr), PGSIZE, V2P(mem), PTE_W | PTE_U) < 0) {
    cprintf("pagefault: mappages failed\n");
    kfree(mem);
    curproc->killed = 1;
    return;
  }

  // 페이지 테이블 로드
  lcr3(V2P(curproc->pgdir));
  cprintf("[Pagefault] Allocate new page at address: 0x%x\n", PGROUNDDOWN(addr));
}
```

mem 변수는 할당된 메모리 페이지의 포인터, pte는 페이지 테이블 엔트리의 포인터이다. 먼저, rcr2()를 통해 CPU의 CR2 레지스터에서 해당 주소를 읽어, 폴트가 발생한 주소가 커널 주소 공간 이상이거나 프로세스의 주소 공간, 또는 스택 주소를 벗어난 경우 유효하지 않은 접근으로 간주하여 프로세스를 종료시킨다. 이후 xv6에 내장된 walkpgdir 함수를 호출하여 프로세스 주소에 해당하는 페이지 테이블 엔트리의 주소를 pte에 저장하고, 만약 엔트리가 존재하지 않는 프로세스일 경우에는 에러를 출력하고 프로세스를 종료한다.

```c
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

walkpgdir func

```c
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

mappages func

물리 페이지 할당의 경우, kalloc()함수를 호출하여 새로운 물리 페이지를 할당받는다. 할당에 실패할 경우에 예외 처리를 구현하였고, memset 함수를 사용하여 새롭게 할당된 메모리를 0으로 초기화한다. 새로운 페이지 매핑은 mappages 함수를 호출하여 새로 할당받은 물리 메모리 mem을 페이지 폴트가 발생한 주소에 연결한다. mapping에 실패할 경우에는 새롭게 할당받는 메모리를 kfree 함수로 해제한 후 프로세스를 종료하게 된다.

```c
int
allocuvm_stack(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    if (a == newsz - PGSIZE || a == newsz - PGSIZE * 5) {
      mem = kalloc();
      if(mem == 0){
        cprintf("allocuvm out of memory\n");
        deallocuvm(pgdir, newsz, oldsz);
        return 0;
      }
      memset(mem, 0, PGSIZE);
      if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
        cprintf("allocuvm out of memory (2)\n");
        deallocuvm(pgdir, newsz, oldsz);
        kfree(mem);
        return 0;
      }
    }
  }
  return newsz;
}
```

페이지 폴트의 구현 확인을 위해 allocuvm 함수를 수정하여 특정 위치에만 메모리를 할당하도록 변경하였고, stack, heap 영역 확장 시 페이지 폴트를 유도하여 폴트 핸들러가 나머지 페이지를 동적으로 할당하도록 설계하였다. 기존의 정적으로 4KB 배수의 페이지를 할당하던 부분에서 메모리 사용의 효율성을 높이고, stack과 heap 영역의 확장에 따라 메모리를 할당하여 HW 자원을 효율적으로 사용할 수 있다.

```c
void test_address(char *addr, char value);

int
main(int argc, char *argv[])
{
  char *valid_addr = sbrk(PGSIZE * 2); // 2 페이지 크기만큼 메모리를 할당
  char *page_fault_addr = valid_addr + (PGSIZE * 3);  // 할당된 메모리 영역 바로 밖의 주소를 설정
  char *invalid_addr = (char*)0x80000000; // 커널 공간의 주소를 사용해 페이지 폴트

  // 유효한 주소 테스트
  printf(1, "Testing valid address:\n");
  test_address(valid_addr, 'V');

  // 페이지 폴트를 유발하는 유효한 주소 테스트
  printf(1, "Testing page fault address:\n");
  test_address(page_fault_addr, 'F');

  // 유효하지 않은 주소 테스트
  printf(1, "Testing invalid address:\n");
  test_address(invalid_addr, 'I');

  exit();
}

void test_address(char *addr, char value)
{
  printf(1, "Accessing address: 0x%x\n", addr);

  *addr = value;

  // 페이지 폴트가 처리되면 실행됨
  printf(1, "Value at address: %c\n", *addr);
  printf(1, "If This line is printed Memory allocate Success.\n\n");
}
```

page fault test를 위해 user program 작성

시나리오는 3가지로 구성하였다. sbrk 함수를 사용하여 2페이지 크기만큼의 메모리를 할당하고, 할당된 메모리 영역 밖의 주소(page_fault_addr)와 커널 영역의 주소를 설정하고 각 주소에 대한 테스트를 시행한다.

valid address가 들어올 경우 주소에 대한 유효성 여부를 출력하고 접근 어드레스에 대한 출력, 메모리 할당 여부를 반환한다. 할당된 메모리 영역 밖의 주소가 접근될 경우 page fault가

발생된 뒤 페이지 폴트 핸들러가 정상적으로 작동하면, 새로운 메모리 영역을 할당하여 유효성에 대한 반환값과 재할당 여부를 출력한다.

커널 공간 주소에 접근될 경우에도 page fault가 발생되나, 페이지 폴트 핸들러가 접근할 수 있는 영역을 벗어난 범위이기 때문에 프로세스가 종료된다.

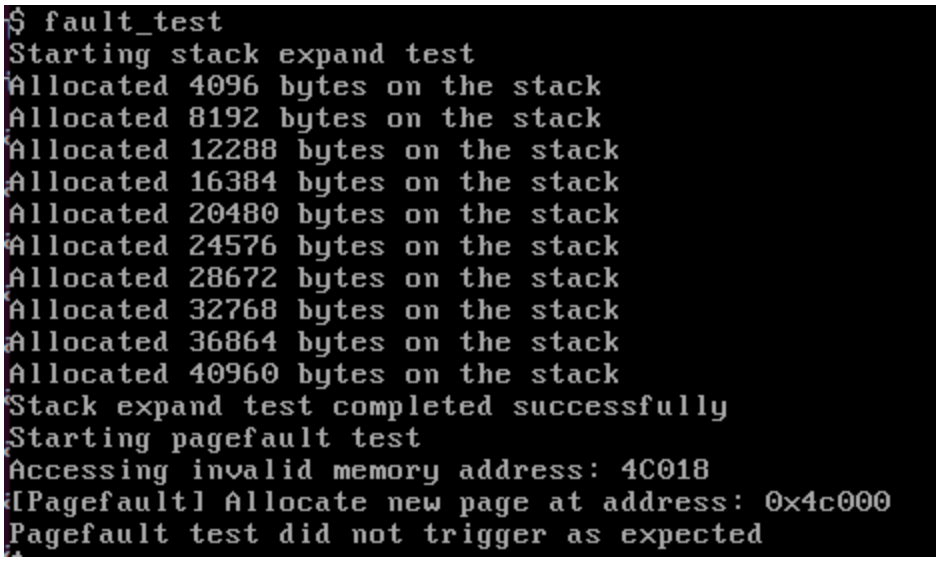
```
$ fault_test2
Testing valid address:
Accessing address: 0x6000
Value at address: U
If This line is printed Memory allocate Success.

Testing page fault address:
Accessing address: 0x9000
[Pagefault] Allocate new page at address: 0x9000
Value at address: F
If This line is printed Memory allocate Success.

Testing invalid address:
Accessing address: 0x80000000
[Pagefault] Invalid access at address: 0x80000000
```

예상 시나리오 대로 동작하는 결과 값을 확인할 수 있었다. “If This line is printed Memory allocate Success”문구에서 확인할 수 있듯이, 메모리 영역이 정상적으로 할당되어 프로세스가 실행될 경우에만 해당 문장이 출력되도록 설계하였다.

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "memlayout.h"

void stack_expand_test() {
    int stack_size = 0;
    char *stack_pointer;

    printf(1, "Starting stack expand test\n");

    // Allocate memory on the stack in small increments to force stack expansion
    for (int i = 0; i < 10; i++) {
        stack_size += 4096; // 4KB
        stack_pointer = (char *)malloc(stack_size);
        if (stack_pointer == 0) {
            printf(1, "Failed to allocate memory on the stack\n");
            exit();
        }
        printf(1, "Allocated %d bytes on the stack\n", stack_size);
    }

    // Access the newly allocated stack memory to force page fault handling
    for (int i = 0; i < stack_size; i += 4096) {
        stack_pointer[i] = 'a';
    }

    printf(1, "Stack expand test completed successfully\n");
}

void pagefault_test() {
    char *valid_addr = sbrk(PGSIZE * 2); // Allocate 2 pages
    char *page_fault_addr = valid_addr + (PGSIZE * 3);  // Address just outside allocated memory

    printf(1, "Starting pagefault test\n");

    // Access invalid memory to trigger a page fault
    printf(1, "Accessing invalid memory address: %p\n", page_fault_addr);
    *page_fault_addr = 'a';

    printf(1, "Pagefault test did not trigger as expected\n");
}

int main(int argc, char *argv[]) {
    stack_expand_test();
    pagefault_test();
    exit();
}
```

fault_test1은 stack을 확장하면서 page fault를 유도하고, 폴트 핸들러가 이를 처리하여 새로운 메모리를 할당하여 값을 쓸 수 있도록 구성한 유저 프로그램이다.

스택의 확장이 일어나면서 새로운 페이지가 할당되는 것을 확인할 수 있고, page fault를 유발하여 새로운 메모리를 할당하는 것을 확인할 수 있다.



5. File System of xv6

3) 기존 xv6 File System

기존 xv6 파일 시스템은 디스크 블록을 기본 단위로 사용하여 파일과 디렉토리를 관리한다. 주요 구성 요소에는 슈퍼블록, inode, 데이터 블록, 디렉토리 엔트리, 비트맵이 실제 xv6에 주요 구조체들로 구성이 되어 있다.

그중 inode는 파일의 메타데이터를 저장하는 구조체인데, 파일의 유형(파일, 디렉토리, 장치 등), 권한, 소유자, 크기, 데이터 블록 포인터 등을 포함하며 inode 번호를 통해 파일을 참조한다.

xv6의 inode 구조체는 NDIRECT 개의 직접 블록(direct block)과 하나의 간접 블록 포인터를 포함한다. 간접 블록 포인터는 추가적인 데이터 블록 포인터를 포함하는 블록을 가리킨다.

// file.h에 속한 inode 구조체

C file.h

```
12  // in-memory copy of an inode
13  struct inode {
14    uint dev;           // Device number
15    uint inum;          // Inode number
16    int ref;            // Reference count
17    struct sleeplock lock; // protects everything below here
18    int valid;          // inode has been read from disk?
19
20    short type;         // copy of disk inode
21    short major;
22    short minor;
23    short nlink;
24    uint size;
25    uint addrs[NDIRECT+1];
26  };
```

xv6의 기본 파일 시스템은 간단한 구조로, 한 번의 간접 블록만을 지원하여 최대 파일 크기를 제한한다.

4) Double indirect 구현

기존 xv6의 inode에서 한 번의 indirect를 지원했던 것에서 확장하여, 더 큰 파일을 다룰 수 있도록 double indirect를 구현하여 더 큰 파일 크기 지원을 목표로 한다.

기존의 xv6 시스템인 direct+single indirect 구조에서 확장하여 double indirect를 구현하는 것이 목표이다. 기존 inode에서는 끝에 addr128의 테이블을 가리키는 구조로 되어있다. 여기에서 direct로 할당된 addr12를 single indirect로 바꿔주고, 마지막 addr13을 double indirect로 바꿔서 총 3번 접근을 통해 data에 접근할 수 있도록 구조를 변경한다. 주로 수정해야 할 함수는 fs.c의 bmap함수로, 인자로 받은 inode상의 bn번째 데이터블록을 가져오는 함수인 bmap함수만 올바르게 지정하도록 수정한다면 나머지 함수는 수정하지 않아도 알아서 double indirect에 맞도록 전체적인 구조가 변경될 것이다. 그 외 데이터 블록을 deallocate하는 함수인 itrunc함수도 double indirect 블록이 있을 때 그에 맞게 deallocate하도록 코드를 추가할 필요가 있으며, 이 때 가장 안쪽의 indirect block부터 차례대로 초기화해줘야 한다. 마지막으로 fs.h, param.h와 inode, dinode 구조체에서 수정사항에 맞춰 상수값을 일부 바꿔주면 된다.

Makefile에 test file을 추가한다.

Makefile

```
171    UPROGS=\
191        _trace\
192        _ps\
193        _fcfs_test\
194        _ml_test\
195        _mlfq_test\
196        _file_test\
```

a. DIRECT 12개+SINGLE INDIRECT 1개 구조에서 DIRECT 11개+SINGLE INDIRECT 1개+DOUBLE INDIRECT 1개 구조로 수정된다. 따라서 fs.h에서 NDIRECT값을 12에서 11로 바꿔주어 11번째부터 SINGLE INDIRECT로 지정한다. 또한 MAXFILE값도 변경되는데, 기존에는 DIRECT 12개+SINGLE INDIRECT에서 128개(BSIZE/sizeof(uint))의 데이터를 가리켰으나, 여기에 DOUBLE INDIRECT에서 추가로 128*128*개의 데이터를 가리키게 되므로 *NINDIRECT*NINDIRECT값을 추가로 더해줘야 한다.

fs.h

```
24    #define NDIRECT 11
25    #define NINDIRECT (BSIZE / sizeof(uint))
26    #define MAXFILE (NDIRECT + NINDIRECT + (NINDIRECT*NINDIRECT))
```

b. 또한 param.h에서 double indirect를 구현해준 것과 맞춰 더 큰 파일 사이즈를 가질 수 있도록 FSSIZE값을 20000으로 늘려주었다.

param.h

```
1    #define NPROC        64  // maximum number of processes
2    #define KSTACKSIZE 4096  // size of per-process kernel stack
3    #define NCPU          8  // maximum number of CPUs
4    #define NOFILE       16  // open files per process
5    #define NFILE       100  // open files per system
6    #define NINODE       50  // maximum number of active i-nodes
7    #define NDEV         10  // maximum major device number
8    #define ROOTDEV       1  // device number of file system root disk
9    #define MAXARG       32  // max exec arguments
10   #define MAXOPBLOCKS  10  // max # of blocks any FS op writes
11   #define LOGSIZE      (MAXOPBLOCKS*3)  // max data blocks in on-disk log
12   #define NBUF         (MAXOPBLOCKS*3)  // size of disk block cache
13   #define FSSIZE       20000  // size of file system in blocks
```

c. 우선 fs.c의 bmap함수에서 Double indirect를 구현하도록 내용을 추가해주어야 한다. 기존에는 bn값이 NDIRECT+NINDIRECT 이상인 경우 if문에 전부 걸리지 않고 panic을 호출하게 되어있었다. 여기에서 panic으로 가기전에 bn값에서 NINDIRECT를 빼주고(single indirect처리 직전에 bn에서 NDIRECT를 빼준 것과 같은 이유), bn이 NINDIRECT*NINDIRECT보다 작게 되면, 즉 double indirect로 감당이 가능한 bn값인 경우 double indirect로 처리하는 코드를 다음과 같이 추가하게 된다. (추가하는 코드에서 필요한 변수를 line 377-378에 추가하고, line 400-426에 double indirect에 대한 코드를 추가하였다.

fs.c

```
372   static uint
373   bmap(struct inode *ip, uint bn)
374   {
375     uint addr, *a;
376     struct buf *bp;
377     uint *a2; // add
378     struct buf *bp2; // add
```

fs.c

```
374   {
387     if(bn < NINDIRECT){
388       // Load indirect block, allocating if necessary.
389       if((addr = ip->addrs[NDIRECT]) == 0)
390         ip->addrs[NDIRECT] = addr = balloc(ip->dev);
391       bp = bread(ip->dev, addr);
392       a = (uint*)bp->data;
393       if((addr = a[bn]) == 0){
394         a[bn] = addr = balloc(ip->dev);
395         log_write(bp);
396       }
397       brelse(bp);
398       return addr;
399     }
400   // add code
401     bn -= NINDIRECT;
402
403     if(bn < NINDIRECT*NINDIRECT) { // second-level indirection
404       if((addr = ip->addrs[NDIRECT+1]) == 0) // double indirect is next to single indirect
405         ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
406       bp = bread(ip->dev, addr);
407       a = (uint*)bp->data;
408
409       // go to (bn/NINDIRECT)th inner table
410       if((addr = a[(bn/NINDIRECT)]) == 0) {
411         a[(bn/NINDIRECT)] = addr = balloc(ip->dev);
412         log_write(bp);
413       }
414       brelse(bp);
```

```
415
416       bp2 = bread(ip->dev, addr);
417       a2 = (uint*)bp2->data; // for inner table
418
419       // go to (bn%NINDIRECT)th entry of (bn/NINDIRECT)th inner table
420       if((addr = a2[(bn%NINDIRECT)]) == 0) {
421         a2[(bn%NINDIRECT)] = addr = balloc(ip->dev);
422       log_write(bp2);
423       }
424       brelse(bp2);
425       return addr;
426     }
427
428     panic("bmap: out of range");
429   }
```

기본적인 구조는 기존에 있던 single indirect 코드에서 확장한 형태이다. 우선 data를 가리키는 addr들을 담고 있는 inner table를 먼저 찾아야 하는데, 이는 line 404-line 414에 구현하였으며 기존 single indirect(line 389-397)부분과 거의 유사한 것을 확인할 수 있다. 차이점은 single indirect의 다음 addr에서 시작한다는 점과 bn/NINDIRECT번째의 inner table을 찾아야 하기 때문에 a인덱스로 해당값이 들어간다는 점이다. (double indirect만 생각했을 때, inner table의 addr1-addr128이 outer table의 addr1에 들어있고, adr129-adr256이 outer table의 addr2에 들어있는 구조이기 때문이다) 그렇게 해서 addr를 찾고 그 정보를 bp2와 a2에 담으면(line 416-417), 여기에는 우리가 찾고자 하는 data의 addr를 담고 있는 inner table의 정보가 들어있게 된다. 여기에서 방금 했던 작업을 다시 한번 반복해서 bn/NINDIRECT번째의 inner table에서 bn%NINDIRECT번째의 entry를 찾아 데이터블록을 가져오게 하면 된다(line 419-425).

d. fs.c의 itrunc함수에서 double indirect 블록이 있을 때 그에 맞게 deallocate하도록 코드를 추가해야 한다. 우선 추가하는 코드에서 필요한 변수를 line 442-443에 추가하였다.

C fs.c

```
436   static void
437   itrunc(struct inode *ip)
438   {
439     int i, j, k;
440     struct buf *bp;
441     uint *a;
442     uint *a2; // add
443     struct buf *bp2; // add
```

기존 single indirect에서는 single indirect의 데이터블록이 있을 경우 이중 루프를 통해서 indirect가 가리키는 데이터 블록들을 먼저 deallocate해주고, 마지막으로 indirect block을 deallocate해주었다. 같은 방식으로 double indirect에서는 삼중 루프를 통해서 inner table->outer table->double indirect block 순으로 deallocate를 해주었다.

fs.c

```
463   // add code
464     if(ip->addrs[NDIRECT+1]) { // second-level indirection
465       bp = bread(ip->dev, ip->addrs[NDIRECT+1]);
466       a = (uint*)bp->data;
467       for(j = 0; j < NINDIRECT; j++) { // bfree a[j]'s entry, then bfree a[j]
468         if(a[j]) {
469           bp2 = bread(ip->dev, a[j]);
470           a2 = (uint*)bp2->data;
471           for(k = 0; k < NINDIRECT; k++) {
472             if(a2[k])
473               bfree(ip->dev, a2[k]);
474           }
475           brelse(bp2);
476           bfree(ip->dev, a[j]);
477           a[j] = 0;
478         }
479       }
480       brelse(bp);
481       bfree(ip->dev, ip->addrs[NDIRECT+1]);
482       ip->addrs[NDIRECT+1] = 0;
483     }
484
485     ip->size = 0;
486     iupdate(ip);
487   }
488
```

e. 마지막으로 변경한 구조에 맞춰 inode, dinode 구조체에 변경사항을 반영해야 한다. 기존 inode, dinode에서는 addrs를 NDIRECT+1개 할당하였으나(direct+single indirect), double indirect를 구현하는 과정에서 NDIRECT값을 하나 빼고 double indirect를 하나 추가하였다. 따라서 addrs배열의 원소를 NDIRECT+2로 수정해야 한다(direct+single indirect+double indirect). inode 구조체는 file.h에, dinode 구조체는 fs.h에 있다는 것을 확인하였고 addrs 배열의 길이만 다음과 같이 수정하였다.

fs.h

```
28  // On-disk inode structure
29  struct dinode {
30    short type;           // File type
31    short major;          // Major device number (T_DEV only)
32    short minor;          // Minor device number (T_DEV only)
33    short nlink;          // Number of links to inode in file system
34    uint size;            // Size of file (bytes)
35    uint addrs[NDIRECT+2];   // Data block addresses +2
36  };
```

file.h

```
10
11
12  // in-memory copy of an inode
13  struct inode {
14    uint dev;           // Device number
15    uint inum;          // Inode number
16    int ref;            // Reference count
17    struct sleeplock lock; // protects everything below here
18    int valid;          // inode has been read from disk?
19
20    short type;         // copy of disk inode
21    short major;
22    short minor;
23    short nlink;
24    uint size;
25    uint addrs[NDIRECT+2];
26  };
```

f. 파일을 모두 위와 같이 변경한 이후 make qemu로 컴파일을 진행하고 xv6에서 file_test 파일을 실행한 결과 정상적으로 나오는 것을 확인할 수 있었다. double indirect를 구현하기 전에 실행했을 때는 파일크기가 너무 커서 정상적으로 진행되지 않고 프로그램이 종료되었으나, double indirect를 구현한 후 실행했더니 다음과 같이 파일을 쓰고 읽는 것을 확인할 수 있었다.

QEMU

Machine View

```
Username: minwoo
Password: 0522
init: starting sh
$ file_test
Test 1: Write 8388608 bytes
Test 1 passed

Test 2: Read 8388608 bytes
Test 2 passed

Test 3: repeating test 1 & 2
Loop 1: 1.. 2.. ok
Loop 2: 1.. 2.. ok
Loop 3: 1.. 2.. ok
Loop 4: 1.. 2.. ok
Loop 5: 1.. 2.. ok
Loop 6: 1.. 2.. ok
Loop 7: 1.. 2.. ok
Loop 8: 1.. 2.. ok
Loop 9: 1.. 2.. ok
Loop 10: 1.. 2.. ok
Test 3 passed
All tests passed!!
$
```

<conclusion>

본 프로젝트에서는 miniOS와 xv6 프로젝트를 통해 운영체제의 기본적이고 고급 기능을 종합적으로 탐구하였다. 셸 명령어, 시스템 호출, CPU 스케줄링, 파일 시스템의 실제 구현을 통해 운영 체제의 핵심 기능을 관리하고 조작하는 실용적인 경험을 하였다.

프로젝트의 결과, 팀원들은 Linux 환경에서 miniOS의 셸 명령 구현 및 xv6 환경에서의 시스템 호출과 CPU 스케줄링 방식의 구현과 파일 시스템(inode의 기존 구조에서 double indirect 구조로 변경) 구현을 성공적으로 수행하였다. 이를 통해, 운영체제의 다양한 기능들이 실제 하드웨어 및 소프트웨어와 어떻게 상호작용하는지에 대한 깊은 이해를 얻을 수 있었다.

xv6의 소스 코드는 간결하고 이해하기 쉬워 커스터마이징이 필요한 부분을 신속하게 파악하고 수정할 수 있었다. 이는 특히 스케줄링 알고리즘과 파일 시스템의 구조 변경에 큰 도움이 되었으며, 커스터마이징 과정에서 발생한 다양한 문제들을 해결하면서 디버깅 기술이 향상되었다.

특히, xv6의 스케줄링 로직과 파일 시스템의 인터페이스를 수정하면서 시스템 호출과 프로세스 상태 관리에 대한 깊은 이해를 얻을 수 있었고, 이러한 경험은 복잡한 시스템을 분석하고 필요한 기능을 설계, 구현하는 능력을 키우는 데 큰 도움이 되었다.

운영체제 과목을 통해 이론적으로 배운 내용을 실제로 적용해 볼 수 있는 기회가 되었으며, 특히 xv6와 같은 교육용 운영체제를 사용함으로써 이론과 실제의 연결고리를 더욱 명확히 할 수 있었다. 프로젝트를 통해 팀원들은 문제 해결 능력을 향상시킬 수 있었고, 운영체제에 대해 깊게 공부했다는 것을 남길 수 있었다.

<Weekly schedule Plan>

	이현종	정재학	김민우	전광현
역할	- 프로젝트 주제 선정 - user program - memory allocation 구현 - 보고서 작성 - 기존 xv6 OS 구조 분석	- system calls - ML scheduling 구현 - Block diagram 작성 - 보고서 작성 - 발표 자료 준비	- Shell commnad 구현 - MLFQ schedule 구현 - File system 구현 - trouble shooing - 발표 자료 준비	-FCFS schedule 구현 - user program - 보고서 작성 - Conclusion - 발표자료 준비
모임 일자	수행 내용			
4/17	**project**의 의의와 구현 가능한 목표 설정 -이현종 : miniOS 내 구현 가능 주제 제안			

	-정재학 : project 목표 정리 -김민우 : 쉘 커맨드 구현 제안 -전광현 : OS 핵심 개념 및 학습 필요성 제안
4/24	**idea** 구상을 위한 **brainstorming** 후 **miniOS shell command** 구현 계획 설정 - 컴퓨터학부 OS 진행 방향 공유
5/1	**git**을 이용한 작업 계획 수립 및 **OS**에 대한 전체적인 조사
5/8	**ls command**와 **cd command** 구현 후 **test** -이현종 : parcing 추가 -정재학 : 코드 debug -김민우 : Shell command 구현 -전광현 : 발표 자료 준비
5/15	**miniOS**에 목표로 했던 모든 **shell command** 구현 **xv6**와 **qemu**를 이용하기로 방향성 변경 qemu 에뮬레이터 필요성과 기존 linux miniOS 구현 비교
5/22	**xv6** 적응을 위한 기초적인 **command** 구현과 **priority scheduling** 구현 후 **test** - xv6 process scheduling 분석 - user program, system calls 구현
5/29	**CPU scheduling**에 대해 **3**가지 방식으로 구현 하고 **file system** 기능 구현에 대한 **idea** 회의 - xv6 file system 분석 - file system 구현 - FCSF, ML, MLFQ scheduling 추가 구현
6/5	**file system, memory** 구현 후 **test,** 그리고 최종 보고서 작성 및 발표 자료 준비 - memory allocation 분석 - page fault 구현
6/12	최종 적인 **OS**에 대한 **test**와 보고서 완성 **-** 발표 자료 준비